

Go and Production Readiness

Daniel Selans

04.18.2019 PDX Golang Meetup

HELLO, I'm Dan

- Started doing Go at New Relic
- Continued doing Go at InVisionApp
- Now doing Go at DigitalOcean
- Prior to doing Go:
 - Worked at data centers doing integration work in Python, Perl, PHP, C#, C++
- Fun fact:
 - Originally from Riga, Latvia
 - Latvian word for: “oh, shit” || “that sucks” || “that’s garbage” in Latvian is “**sviests**”
 - .. which translates to “butter”

What is production readiness?

A: Is your software ready for production traffic?

Why are we talking about PR?

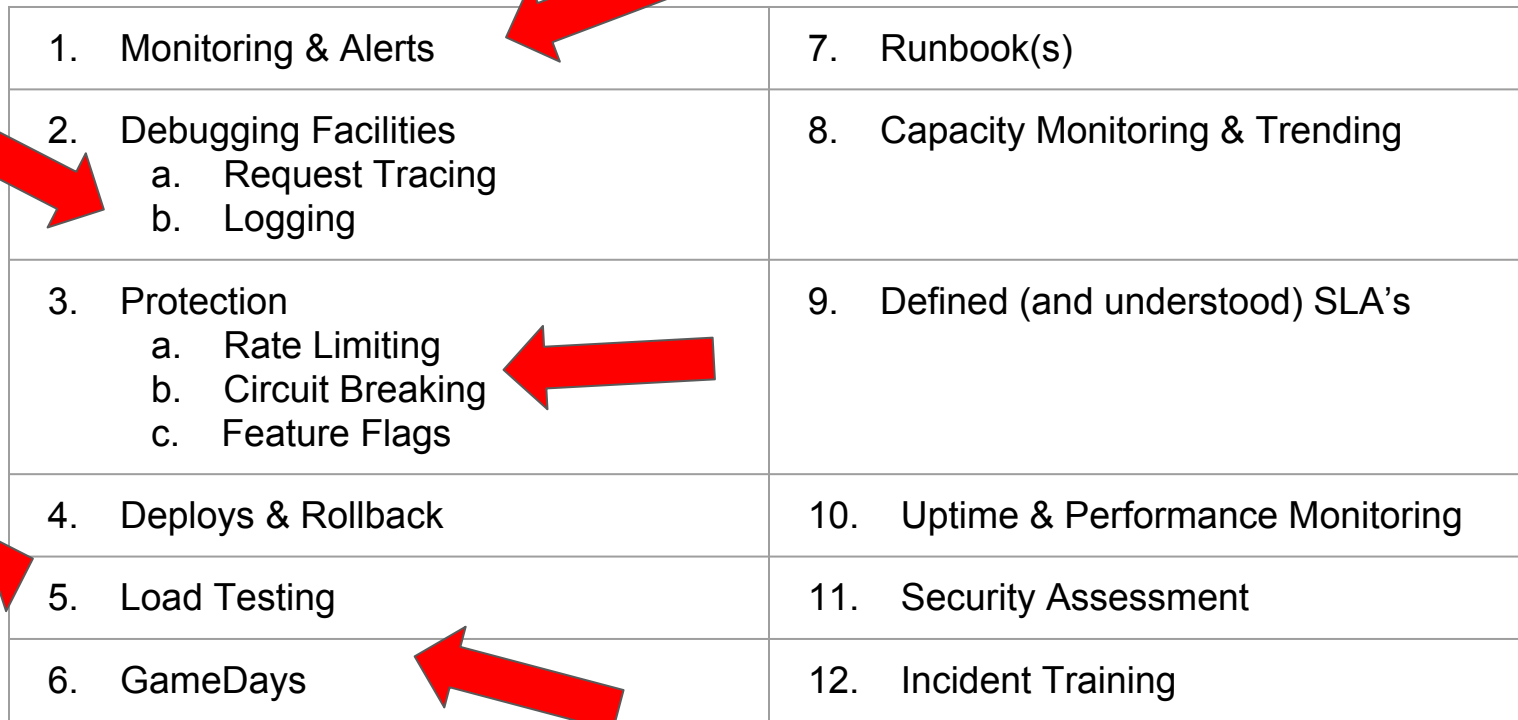
Story time...



What does production readiness consist of?

1. Monitoring & Alerts	7. Runbook(s)
2. Debugging Facilities <ul style="list-style-type: none">a. Request Tracingb. Logging	8. Capacity Monitoring & Trending
3. Protection <ul style="list-style-type: none">a. Rate Limitingb. Circuit Breakingc. Feature Flags	9. Defined (and understood) SLA's
4. Deploys & Rollback	10. Uptime & Performance Monitoring
5. Load Testing	11. Security Assessment
6. GameDays	12. Incident Training

What does production readiness consist of?



1. Monitoring & Alerts	7. Runbook(s)
2. Debugging Facilities <ul style="list-style-type: none">a. Request Tracingb. Logging	8. Capacity Monitoring & Trending
3. Protection <ul style="list-style-type: none">a. Rate Limitingb. Circuit Breakingc. Feature Flags	9. Defined (and understood) SLA's
4. Deploys & Rollback	10. Uptime & Performance Monitoring
5. Load Testing	11. Security Assessment
6. GameDays	12. Incident Training

1. Monitoring & Alerts

- Alert when service is unavailable
- Alert when service is emitting a larger than usual number of errors, 4xx, 5xx
- Alert when service resp time is higher than usual
- And so on...
- Lots of options in this space:
 - APM vendors such as New Relic, AppD, SignalFX, Lightstep, etc.
 - Data analytics such as Datadog
 - Error tracking such as BugSnag, Rollbar
 - Or roll your own:
 - statsd / graphite / grafana / influx / prometheus (time series, metrics-based)
 - nagios, icinga, zabbix, zenoss (traditional monitoring)
 - sentry (error tracking)

1. Monitoring & Alerts

APM in Go looks something like this:

New Relic Example

```
// Instantiate router
routes := mux.NewRouter().StrictSlash( value: true)

// Configure agent
cfg := newrelic.NewConfig( appname: "example-app", license: "license")
cfg.DistributedTracer.Enabled = true

// Instantiate agent
app, _ := newrelic.NewApplication(cfg)

exampleHandler := func(rw http.ResponseWriter, r *http.Request) {
    rw.Write([]byte("Hello!"))
}

// Wrap handler
routes.HandleFunc(newrelic.WrapHandleFunc(app, pattern: "/example", exampleHandler))

// Start server
http.ListenAndServe( addr: ":80", routes)
```

Statsd example

```
func main() {
    // github.com/cactus/go-statsd-client/statsd
    c, _ := statsd.NewClient( addr: "127.0.0.1:8125", prefix: "foo-service")

    // Instantiate router
    routes := mux.NewRouter().StrictSlash( value: true)

    exampleHandler := func(rw http.ResponseWriter, r *http.Request) {
        statName := fmt.Sprintf( format: "request.example.get.200")

        // Time the handler
        defer func(start time.Time) {
            c.TimingDuration(statName + ".duration", time.Now().Sub(start), 0)
        }(time.Now())

        rw.Write([]byte("Hello!"))

        // Inc num success
        c.Inc(statName, 1, 0)
    }

    routes.HandleFunc( path: "/example", exampleHandler)

    // Start server
    http.ListenAndServe( addr: ":8080", routes)
}
```

1. Monitoring & Alerts

Error tracking in Go looks something like this:

BugSnag

```
func main() {
    bugsnag.Configure(bugsnag.Configuration{APIKey: "api-key"})

    http.HandleFunc( pattern: "/unhandled", unhandledCrash)
    http.HandleFunc( pattern: "/handled", handledError)

    http.ListenAndServe( addr: ":9001", bugsnag.Handler( h: nil))
}

func unhandledCrash(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader( statusCode: 200)
    w.Write( []byte("OK\n"))

    // Invalid type assertion, will panic
    func(a interface{}) string { return a.(string) }(struct{}{})
}

func handledError(w http.ResponseWriter, r *http.Request) {
    _, err := os.Open( name: "nonexistent_file.txt")
    if err != nil {
        bugsnag.Notify(err, r.Context())
    }
}
```

Sentry

```
// github.com/getsentry/raven-go
func init() {
    raven.SetDSN("https://<key>:<secret>@sentry.io/<project>")
    raven.SetRelease("optional-release")
    raven.SetIgnoreErrors("some error we want to ignore", "partial error")
}

func main() {
    routes := mux.NewRouter().StrictSlash(true)

    exampleHandler := func(rw http.ResponseWriter, r *http.Request) {
        // Invalid type assertion, will panic
        func(a interface{}) string { return a.(string) }(struct{}{})
    }

    exampleHandler2 := func(rw http.ResponseWriter, r *http.Request) {
        rw.WriteHeader(http.StatusInternalServerError)
        raven.CaptureError(errors.New("something broke"), map[string]string{"tagName": "tagValue"})
    }

    routes.HandleFunc("/example1", raven.RecoveryHandler(exampleHandler))
    routes.HandleFunc("/example2", raven.RecoveryHandler(exampleHandler2))

    // Start server
    http.ListenAndServe(":8080", routes)
}
```

2. Debugging Facilities

A: Are you able to effectively identify problems in your app?

2a. Debugging Facilities: Request Tracing

IF you're doing microservices (or SOA)

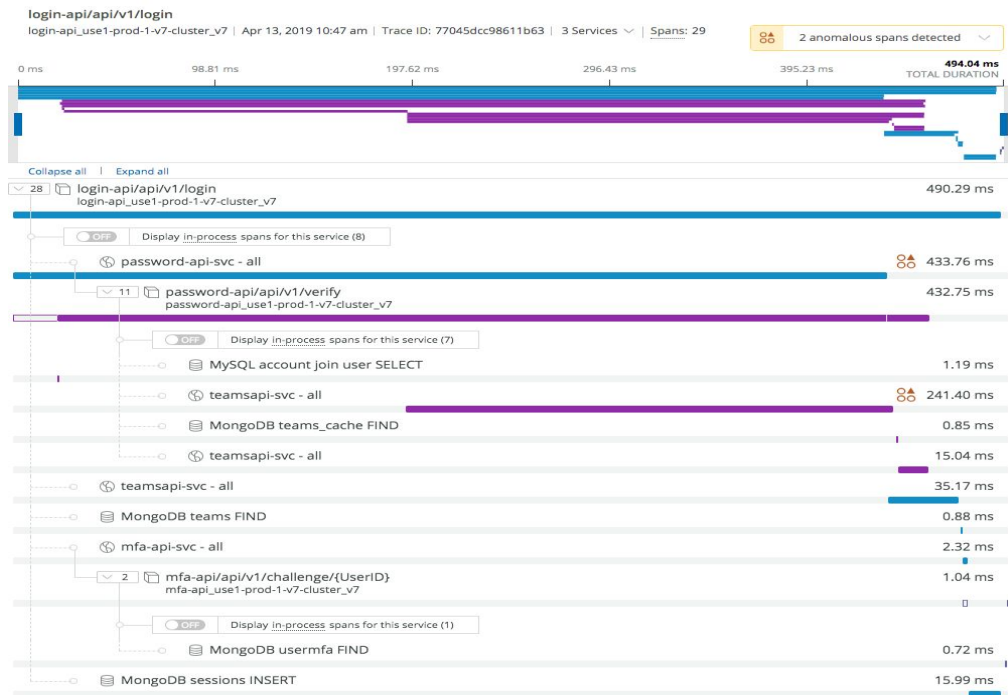
AND you have 2+ dependencies on other services

YOU SHOULD USE REQUEST TRACING

2a. Debugging Facilities: Request Tracing

OK, you don't *need* it, but it'll dramatically improve your quality of life

Here's what
request tracing
looks like in NR:



2a. Debugging Facilities: Request Tracing

There's a bunch of options out there for distributed request tracing both free (Jaeger) and non-free (Lightstep, New Relic, Datadog). Ideally, you want to instrument your code using the open tracing API which will enable you to use many different platforms*.

```
package main

import (
    "context"
    "fmt"
    "github.com/opentracing/opentracing-go"
    "github.com/uber/jaeger-client-go"
    "github.com/uber/jaeger-client-go/config"
)

func printHello(ctx context.Context, name string) {
    span, _ := opentracing.StartSpanFromContext(ctx, operationName: "printHello")
    defer span.Finish()

    span.LogKV(
        alternatingKeyValues...: "event", "fmt.Println")
    span.SetTag(
        key: "name", name)

    fmt.Println(
        a...: "Hello, " + name + "!!")
}

func main() {
    cfg := &config.Configuration{
        Sampler: &config.SamplerConfig{
            Type: "const",
            Param: 1,
        },
        Reporter: &config.ReporterConfig{
            LogSpans: true,
        },
    }

    tracer, closer, _ := cfg.New(
        serviceName: "example-svc",
        config.Logger(jaeger.StdLogger))
    defer closer.Close()

    opentracing.SetGlobalTracer(tracer)

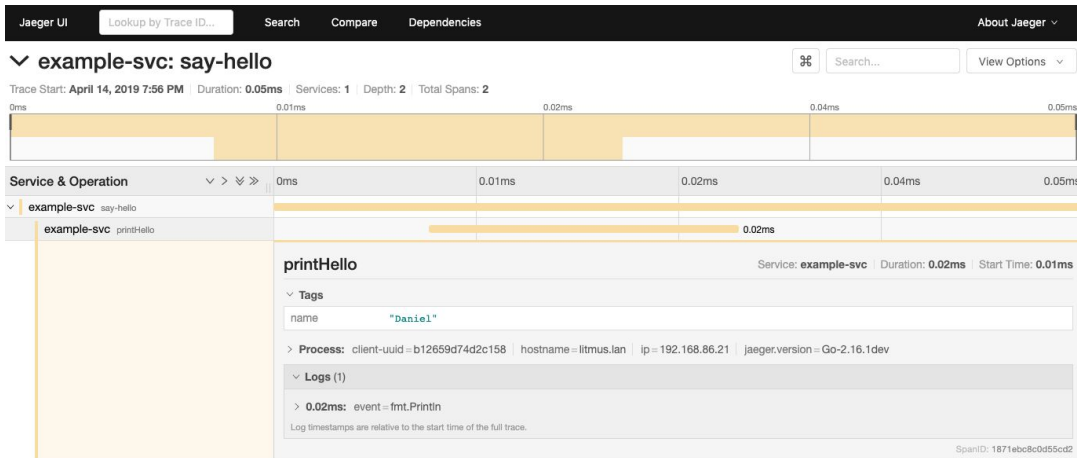
    ctx := context.Background()

    span := tracer.StartSpan(
        operationName: "say-hello")
    defer span.Finish()

    ctx = opentracing.ContextWithSpan(ctx, span)

    printHello(ctx,
        name: "Daniel")
}
```

Example jaeger + open tracing



* Sadly, New Relic's distributed tracing != open tracing :(

2b. Debugging Facilities: Logging

- **Come up with best practices for logging**
 - Even if it's just **"DON'T LOG AS DEBUG IN PRODUCTION"**
- Send only *actionable* logs
- Use a logging vendor (if possible)
 - Logging is a **gigantic** pain in the ass
- Use a logging library that supports structured logging
- I like:
 - uber-go/zap
 - sirupsen/logrus

```
logger.Infof("failed to fetch URL",
    // Structured context as loosely typed key-value pairs.
    "url", url,
    "attempt", 3,
    "backoff", time.Second,
)
```

3a. Protection: Rate Limiting

Ideally, rely on another higher level construct to provide rate limiting protection (such as an API gateway or service mesh)

- Otherwise, use a rate limit middleware for your Go router flavor
 - I've had success with github.com/didip/tollbooth

```
// Instantiate router
routes := mux.NewRouter().StrictSlash( value: true)

// Limit 1 req/sec (per IP)
limit := tollbooth.NewLimiter( max: 1,  tbOptions: nil)

routes.Handle( path: "/example", tollbooth.LimitFuncHandler(limit, func(rw http.ResponseWriter, r *http.Request) {
    rw.Write([]byte("Hi there!"))
}))
```


3b. Protection: Circuit Breakers

Paraphrasing Wikipedia: “... a design pattern that is used for detecting failures and encapsulates the logic of preventing a failure from constantly recurring, during maintenance, temporary external system failure or unexpected system difficulties.”

- What it means: wrap your calls that reach out to external systems with a CB lib
- CB library will keep track of successful and failed requests
- CB library will **fail fast** if calls result in too many consecutive failures
 - This gives the remote systems time to recover AND improves the experience for your users*
 - .. as they no longer have to wait 30s for a timeout to occur

3b. Protection: Circuit Breakers

```
import (
    "context"
    "errors"
    "fmt"
    "github.com/cep21/circuit"
    "github.com/cep21/circuit/closers/hystrix"
    "time"
)

func main() {
    cfg := hystrix.Factory{
        ConfigureOpener: hystrix.ConfigureOpener{
            RequestVolumeThreshold: 5,
        },
        ConfigureCloser: hystrix.ConfigureCloser{
            SleepWindow: 1 * time.Second,
            HalfOpenAttempts: 5,
            RequiredConcurrentSuccessful: 1,
        },
    }

    h := circuit.Manager{
        DefaultCircuitProperties: []circuit.CommandPropertiesConstructor{cfg.Configure},
    }

    c := h.MustCreateCircuit( name: "hello-world")

    for i := 0; i < 30; i++ {
        start := time.Now()

        time.Sleep(100 * time.Millisecond)

        errResult := c.Execute(context.Background(), func(ctx context.Context) error {
            if i > 4 && i < 16 { // after request #10, they should fail faster than 1s
                time.Sleep(time.Second)
                return errors.New( text: "temporary problem")
            }

            return nil
        }, fallbackFunc: nil)

        fmt.Printf( format: "%d: Result: %v Duration: %v\n", i, errResult, time.Now().Sub(start))
    }
}
```

```
litmus:pr dselans$ go run main.go
#0: Result: <nil> Duration: 103.318343ms
#1: Result: <nil> Duration: 103.707043ms
#2: Result: <nil> Duration: 101.774372ms
#3: Result: <nil> Duration: 100.156848ms
#4: Result: <nil> Duration: 103.362037ms
#5: Result: temporary problem Duration: 1.103907982s
#6: Result: temporary problem Duration: 1.103968839s
#7: Result: temporary problem Duration: 1.104776169s
#8: Result: temporary problem Duration: 1.100834277s
#9: Result: temporary problem Duration: 1.104849632s
#10: Result: circuit is open: concurrencyReached=false circuitOpen=true Duration: 103.003203ms
#11: Result: circuit is open: concurrencyReached=false circuitOpen=true Duration: 103.272524ms
#12: Result: circuit is open: concurrencyReached=false circuitOpen=true Duration: 105.081137ms
#13: Result: circuit is open: concurrencyReached=false circuitOpen=true Duration: 100.592074ms
#14: Result: circuit is open: concurrencyReached=false circuitOpen=true Duration: 100.460828ms
#15: Result: circuit is open: concurrencyReached=false circuitOpen=true Duration: 102.002334ms
#16: Result: circuit is open: concurrencyReached=false circuitOpen=true Duration: 103.415701ms
#17: Result: circuit is open: concurrencyReached=false circuitOpen=true Duration: 102.481679ms
#18: Result: circuit is open: concurrencyReached=false circuitOpen=true Duration: 100.368971ms
#19: Result: <nil> Duration: 102.772046ms
#20: Result: <nil> Duration: 102.779697ms
#21: Result: <nil> Duration: 101.044732ms
#22: Result: <nil> Duration: 102.942669ms
#23: Result: <nil> Duration: 105.081298ms
#24: Result: <nil> Duration: 100.973767ms
#25: Result: <nil> Duration: 101.752424ms
#26: Result: <nil> Duration: 100.293156ms
#27: Result: <nil> Duration: 102.230479ms
#28: Result: <nil> Duration: 100.581204ms
#29: Result: <nil> Duration: 100.064357ms
```

3b. Protection: Circuit Breakers

- <https://github.com/sony/gobreaker>
 - First? Simple and straightforward
 - .. maybe too simple
- <https://github.com/rubyist/circuitbreaker>
 - Lots of folks use this
 - .. but no longer maintained :(
- <https://github.com/cep21/circuit>
 - Hystrix-like, feature-complete and actively maintained
 - Significantly more complex than the other libs

3c. Protection: Feature Flags

Short version: Boolean as a service

- Lots of SaaS offerings
 - LaunchDarkly
 - Rollout.io
 - FeatureFlow.io
- If you require a complex rollout strategy - **BUY**
- If you are sure that your business will never want a rollout strategy that covers 2% users from Australia every other Saturday from 2AM - 6AM PST - probably still **BUY**... but you can try **BUILD**

```
func main() {
    client, _ := ld.MakeClient( sdkKey: "YOUR_SDK_KEY", 5*time.Second)

    key := "bob@example.com"
    first := "Bob"
    last := "Loblaw"
    custom := map[string]interface{}{"groups": "beta_testers"}

    user := ld.User{Key: &key,
        FirstName: &first,
        LastName: &last,
        Custom: &custom,
    }

    showFeature, err := client.BoolVariation( key: "YOUR_FEATURE_FLAG_KEY", user, defaultVal: false)
    if err != nil {
        fmt.Println(err.Error())
    }

    if showFeature {
        fmt.Println( a...: "Showing your feature to " + *user.Key)
    } else {
        fmt.Println( a...: "Not showing your feature to " + *user.Key)
    }

    client.Close()
}
```

4. Deploys and Rollbacks

- Perform a deploy and rollback
 - Document it (README.md or your main docs location)
- Note how long deploys and rollbacks take
- Add a `/version` endpoint; populate it at build time via:
 - `RELEASE_VER=$(git rev-parse --short HEAD)`
 - `G00S=linux go build -a -installsuffix cgo -ldflags "-X main.version=$(RELEASE_VER)" -o svc .`



5. Load Testing

*A: Determine the maximum capacity of your service
(before it falls over)*

5. Load Testing

- Things that work well
 - `vegeta` -- high RPS; difficult to implement complex scenarios; Go
 - `locust` -- low RPS; easy to implement complex scenarios; Python
- “Dumb” load testing tools
 - `wrk`, `hey`, `ab`
- A note on taurus:
 - Provides abstracted DSL that works with multiple testing frameworks incl. Locust, JMeter, gatling, grinder and selenium
 - Reality: JMeter works best; others are not fully implemented
 - Blazemeter integration == pretty sweet but \$\$\$

5. Load Testing: Tips

1. Local load testing is better than *NO* load testing
 - a. Avoid overhead by running your application and its dependencies *natively* (outside of Docker)
2. Incorporate local load testing as part of your dev cycle
 - a. Run load test BEFORE and AFTER you have made changes
 - b. Use something simple like `vegeta` or `hey`
3. Ensure GOMAXPROCS is set appropriately
 - a. Your service may be limited to 20% of a single core in K8S but Go will see however many CPU's `sched_getaffinity()` returns and set that as the upper bound for parallel threads
 - b. <https://github.com/uber-go/automaxprocs/>
4. Ideally, load test after every release (as part of CI)
5. Load testing is error prone - if results look weird, re-run & re-verify
6. Graph and analyze CPU and mem metrics during load tests!



6. GameDays

A: Organized Chaos Engineering

6. GameDays

- Paid, proper solutions like Gremlin
 - Awesome, but takes some effort + \$\$\$
 - Will require cross-team collab if you go beyond application layer fault injection
- ... or do it yourself:
 - Do it locally first
 - Come up with failure scenarios
 - Introduce network issues (via pumba)
 - Take down dependencies
 - Write down expectations
 - Run gameday
 - Write down results
 - Rinse, repeat in a group setting in non-local environment

6. GameDays: Local Example

Gameday - login-api - 11.20.2018

Date: 11.20.2018

Participants: [Daniel Selans \(Deactivated\)](#) – Edge Team

Environment: Local

Recording: N/A

Tickets Created: ☒ **EDGE-1934 - Determine circuit breaker strategy** **WON'T DO**

☒ **EDGE-1935 - login-api: Healthchecking is slow (and is semi-synchronous)** **DONE**

☒ **EDGE-1936 - login-api: Slow login failures during DB failure** **DONE** ☒ **EDGE-1959 - login-api: Gameday mongo** **DONE**

Scenarios

Unavailable dependencies on start

Start the service with each dependency made unavailable (password-api, mfa-api, teams-api, users-api, accounts-api, primary store (mongo)).

Expected Result

- Service should NOT be able to start w/o the DB backend
- Service should be able to start w/o other healthy dependencies

Actual Result

1. As expected, the service **was not able to start** w/o its mongo backend.
2. As expected, the service **was able** to start w/o other dependencies.

My (Stacked) Top 5

1. **Good APM solution**
2. **Good logging solution**
 - a. & distributed tracing IF APM does not include it
3. **Test deploys & rollbacks**
4. **Load testing story**
5. ***Documentation***

Results

- The production readiness initiative was **really successful**
- 70+ engineer involvement
- More than 40 services and applications evaluated
- Found many **critical** issues before they were seen by users

- ☐ Alerts
- ☐ Capacity Monitoring/Trending
- ☐ Configuration
- ☐ Debugging facility
- ☐ Deploys and Rollbacks tested
- ☐ Feature Flags
- ☐ Gameday testing and remediation
- ☐ Incident Commander certification
- ☐ Load testing
- ☐ Logging
- ☐ Runbooks
- ☐ Security assessment
- ☐ SLAs defined and reported
- ☐ Status Check
- ☐ [Supporting Components](#)
- ☐ Uptime/Readiness/Performance Monitoring

Alerts

Getting notified when something needs human intervention, or adverse conditions exist that require attention.

- ☐ Do you have an on-call rotation including both a primary and secondary on-call for your service? Link: ____
- ☐ Do you have a defined escalation path? Link: ____
- ☐ Have you tested your escalation path and alerting mechanisms?
- ☐ Are on call members appropriately trained in the operation of the service and common debugging methods?
- ☐ Alerts conform to [our established best practices](#)?
- ☐ Are you alerting on the [golden signals](#) per REQ003?
- ☐ Do you have monitoring sufficient that you are alerted before Support for incident level issues?

Capacity Monitoring

Monitoring the free capacity available to your application. Even if the platform is able to scale your number of instances or memory and CPU limits for you, you must understand and be able to reckon with the CPU, memory, and network capacity required by your application under normal workloads, and follow that usage trend over time.

- ☐ Do you know how much free capacity you have (i.e memory, CPU, disk IO, disk space)? Link: ____
- ☐ Under normal circumstances will you know when to request additional resources before it's too late? Link: ____
- ☐ If you have auto-scaling enabled, will you know if your application is suddenly using more resources than is reasonable?
- ☐ As you near the edge of your capacity, will you be alerted, even if performance does not first degrade?

Configuration

Each service repo contains files that describe to the platform how the service is to be handled.

- ☐ `in-repo.yaml` required sections filled out according to REQ009

Debugging facility

The application has the necessary facilities to identify what is wrong when it breaks. This item is about your ability to identify what's wrong. Logging may or may not be part of your debugging facility. It may also include distributed tracing, error reporting, a feature flag to turn on more



Thx!

github.com/dselans